Ordenação Sofisticada & Linear

Sergio Canuto
sergio.canuto@ifg.edu.br

Algoritmos sofisticados de Ordenação - Merge Sort

- O algoritmo merge sort, também conhecido como ordenação por “intercalação”, é um algoritmo recursivo que usa a ideia de *dividir para conquistar* para ordenar os dados de um array.
- Este algoritmo parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos. Sendo assim, o algoritmo divide os dados em conjuntos cada vez menores para depois ordená-los e combiná-los por meio de intercalação (merge).
- O algoritmo merge sort divide, recursivamente, o array em duas partes, até que cada posição dele seja considerada um array de um único elemento. Em seguida, o algoritmo combina dois arrays de forma a obter um array maior e ordenado. Essa combinação dos arrays é feita intercalando seus elementos de acordo com o sentido da ordenação (crescente ou decrescente). O processo se repete até que exista apenas um array.
- Von Neumann - 1948

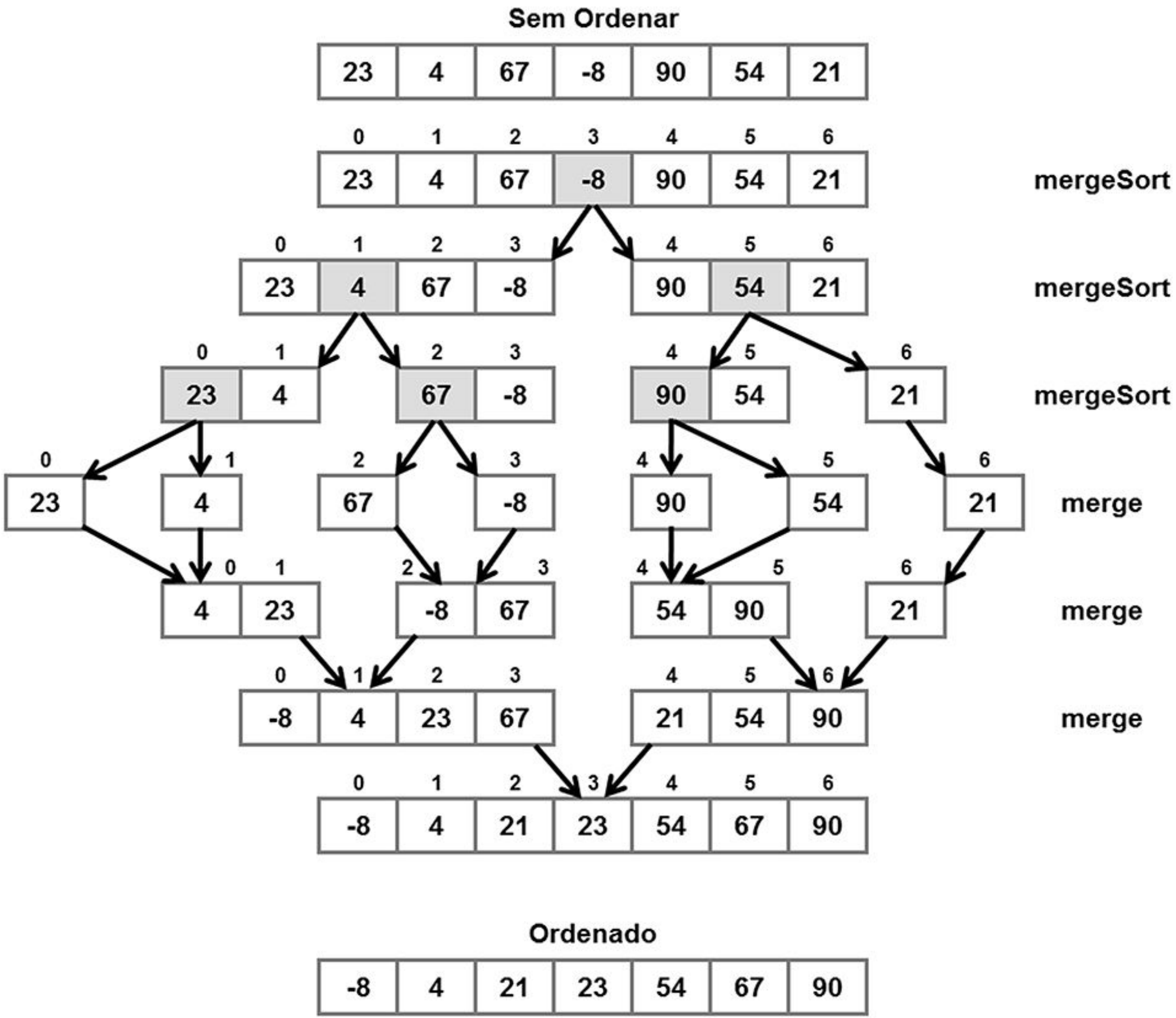
Método merge sort

```c
void merge(int *V, int inicio, int meio, int fim){
    int *temp, p1, p2, tamanho, i, j, k;
    int fim1 = 0, fim2 = 0;
    tamanho = fim-inicio+1;
    p1 = inicio;
    p2 = meio+1;
    temp = (int *) malloc(tamanho*sizeof(int));
    if(temp != NULL){
        for(i=0; i<tamanho; i++){
            if(!fim1 && !fim2){
                if(V[p1] < V[p2])
                    temp[i]=V[p1++];
                else
                    temp[i]=V[p2++];

                if(p1>meio) fim1=1;
                if(p2>fim) fim2=1;
            }else{
                if(!fim1)
                    temp[i]=V[p1++];
                else
                    temp[i]=V[p2++];
            }
        }
        for(j=0, k=inicio; j<tamanho; j++, k++)
            V[k]=temp[j];
    }
    free(temp);
}

void mergeSort(int *V, int inicio, int fim){
    int meio;
    if(inicio < fim){
        meio = floor((inicio+fim)/2);
        mergeSort(V,inicio,meio);
        mergeSort(V,meio+1,fim);
        merge(V,inicio,meio,fim);
    }
}
```

https://scanuto.com/mergesort.c


Sem Ordenar
23 4 67 -8 90 54 21
0 1 2 3 4 5 6
23 4 67 -8 90 54 21
mergeSort
0 1 2 3
23 4 67 -8
4 5 6
90 54 21
mergeSort
0 1
23 4
2 3
67 -8
4 5
90 54
6
21
mergeSort
0
23
1
4
2
67
3
-8
4
90
5
54
6
21
merge
0 1
4 23
2 3
-8 67
4 5
54 90
6
21
merge
0 1 2 3
-8 4 23 67
4 5 6
21 54 90
merge
0 1 2 3 4 5 6
-8 4 21 23 54 67 90
Ordenado
-8 4 21 23 54 67 90

Algoritmos sofisticados de Ordenação - Merge Sort

- Considerando um array com **N** elementos, o tempo de execução do merge sort é sempre de ordem $O(N \log N)$.
  - A eficiência do merge sort não depende da ordem inicial dos elementos.
  - $$T(n) = \begin{cases} O(1), & \text{se } n = 1 \\ 2T(n/2) + O(n), & \text{se } n > 1 \end{cases}$$
- Embora a eficiência do merge sort seja a mesma independente da ordem dos elementos, ele possui um gasto extra de espaço de memória em relação aos demais métodos de ordenação.
- É um algoritmo estável

Algoritmos sofisticados de Ordenação - QuickSort

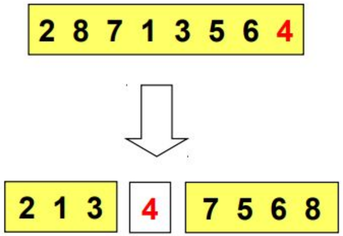
- O algoritmo quick sort, também conhecido como ordenação por “partição”, é outro algoritmo recursivo que usa a ideia de *dividir para conquistar* para ordenar os dados de um array.
- Remaneja um array de modo que os valores menores que certo valor, chamado **pivô**, fiquem na parte esquerda do array, enquanto os valores maiores do que o **pivô** ficam na parte direita.
- Trata-se, em geral, de um algoritmo muito rápido, pois parte do princípio de que é mais fácil ordenar um conjunto com poucos dados do que um conjunto com muitos.
- É um algoritmo lento em alguns casos especiais.
- Princípios:
  - Dividir para Conquistar
  - Ordenar independendemente os problemas menores.
  - Combinar os resultados para produzir a solução do problema maior.

Algoritmos sofisticados de Ordenação - QuickSort

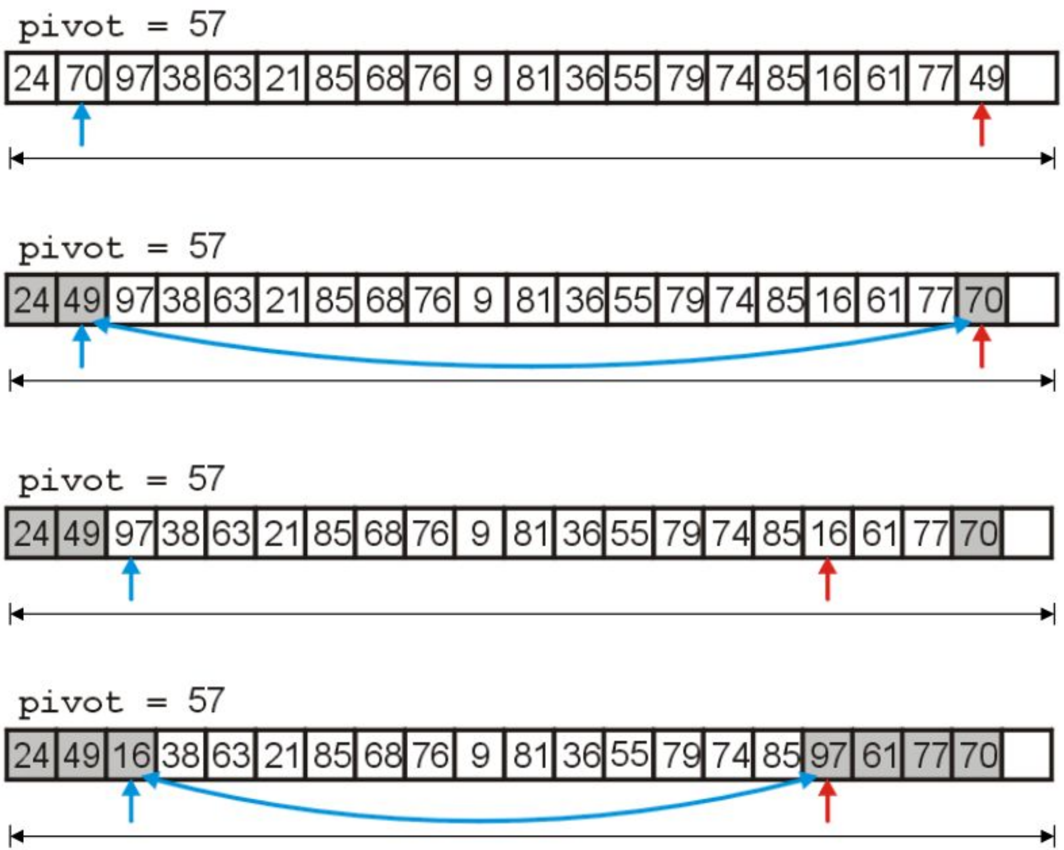
- **A parte mais delicada desse método se refere à divisão da partição**
- **Deve-se rearranjar o vetor na forma A[Esq..Dir] através da escolha arbitrária de um item x do vetor chamado pivô:**
- **Ao final, o vetor A deverá ter duas partes, uma esquerda com chaves menores ou iguais que x e a direita com valores de chaves maiores ou iguais que x:**

✓ A[Esq], A[Esq+1],..., A[j] $\leq$ x

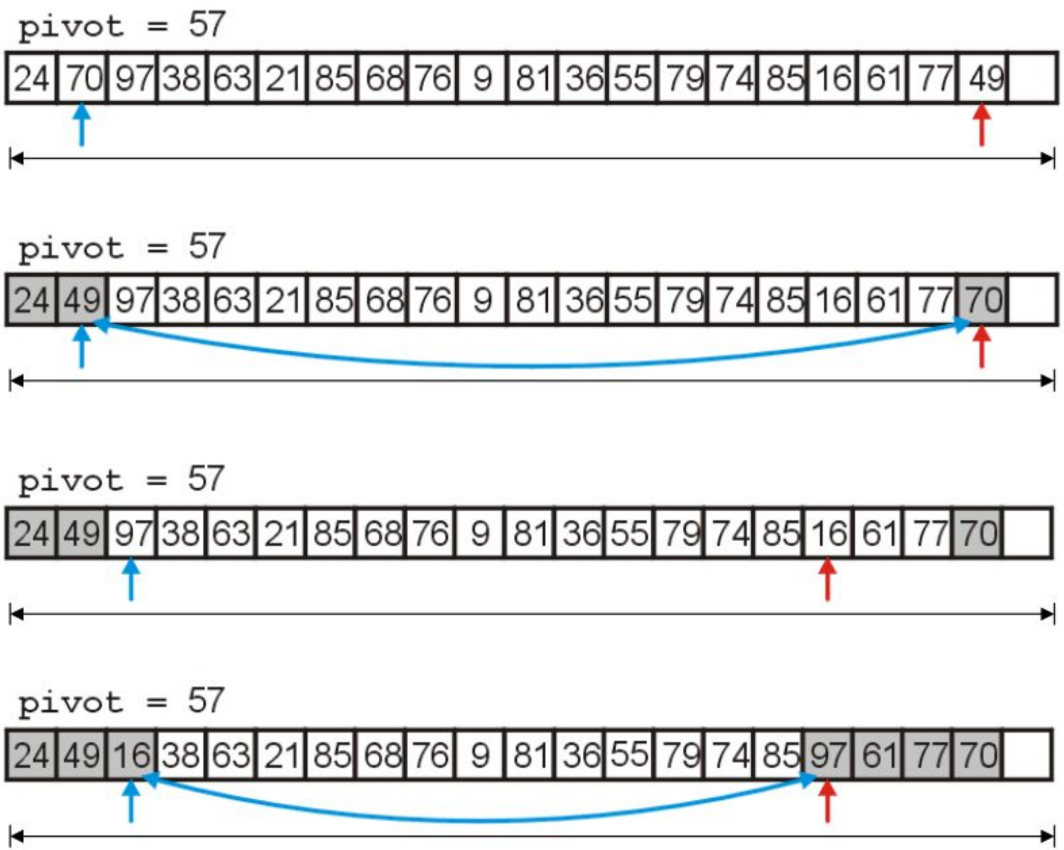
✓ A[i], A[i+1],..., A[Dir] $\geq$ x



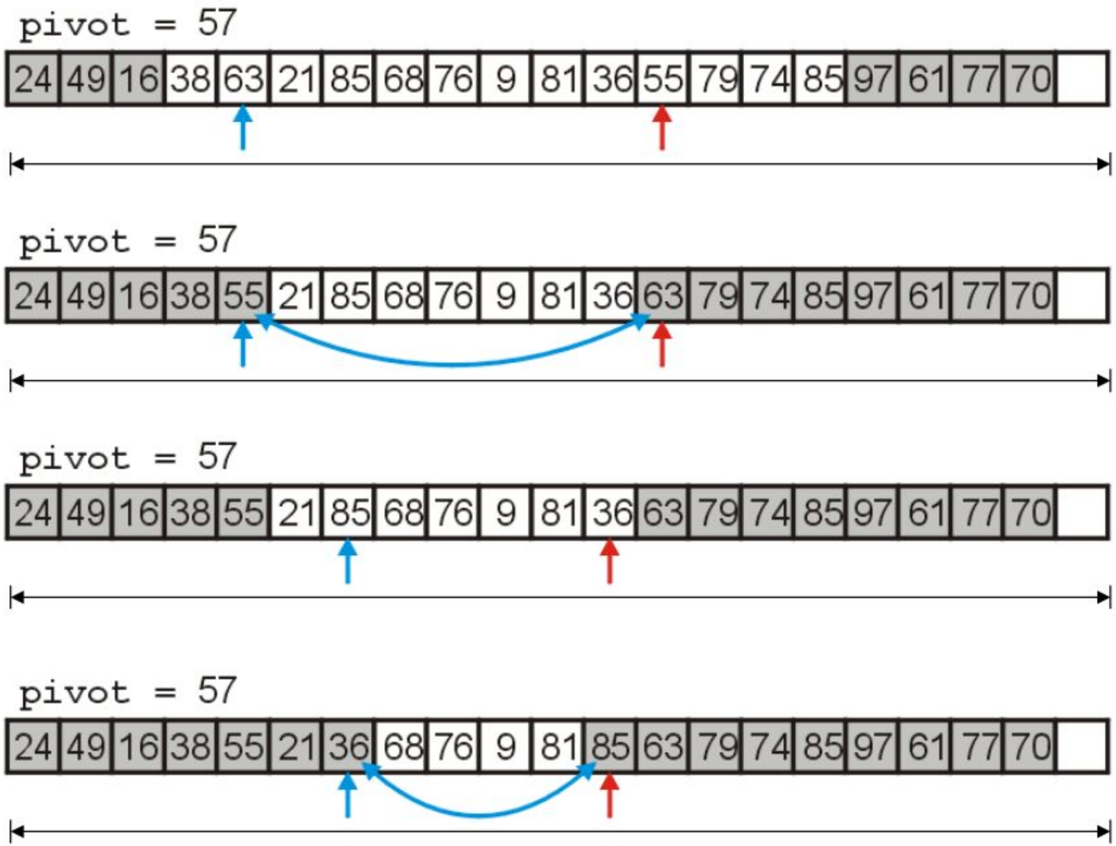
QuickSort - Partição (exemplo)



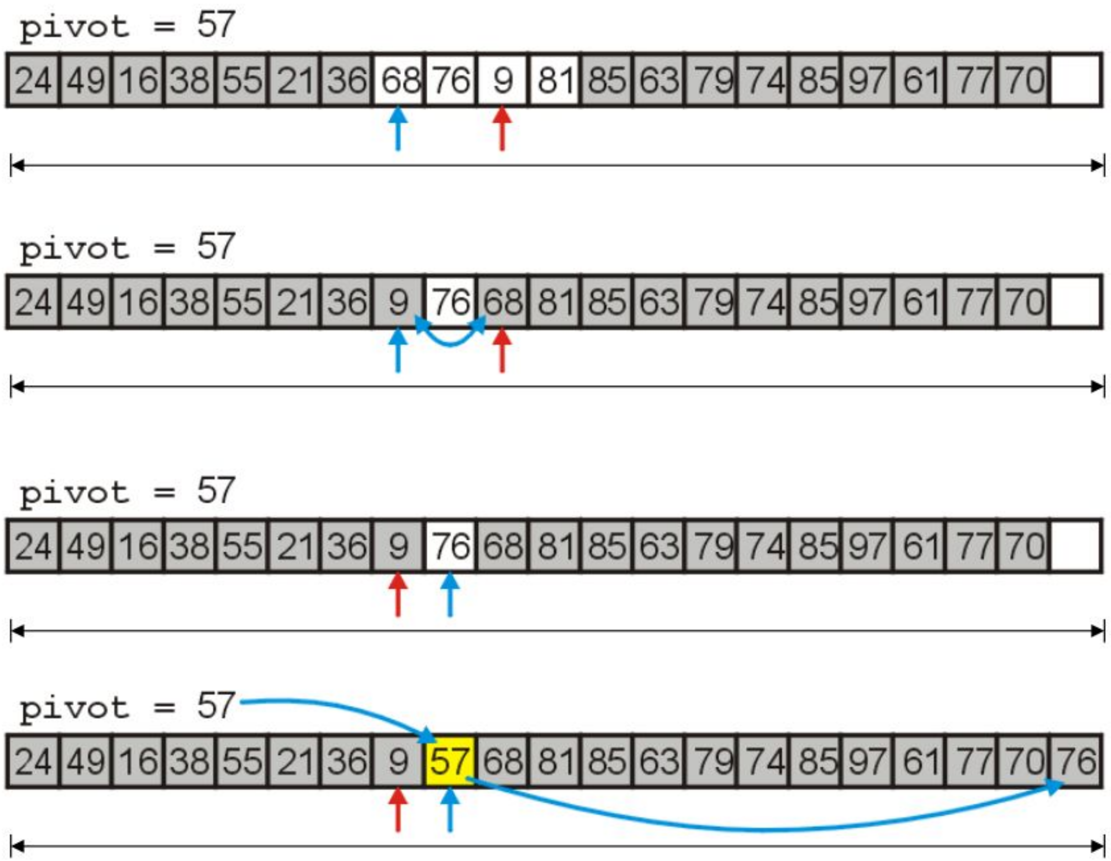
QuickSort - Partição (exemplo)



QuickSort - Partição (exemplo)



QuickSort - Partição (exemplo)



QuickSort - Procedeimento

- Procedimento QuickSort:
  - Escolher arbitrariamente um item x (pivô) do vetor
  - Percorrer o vetor a partir da esquerda até que um item $A[i] \geq x$ é encontrado; da mesma maneira, percorrer o vetor a partir da direita até que um item $A[j] \leq x$ é encontrado;
  - Como os itens A[i] e A[j] não estão na ordem correta no vetor final, eles devem ser trocados
  - Continuar o processo até que os índice i e j se cruzem em algum ponto do vetor

QuickSort - exemplo

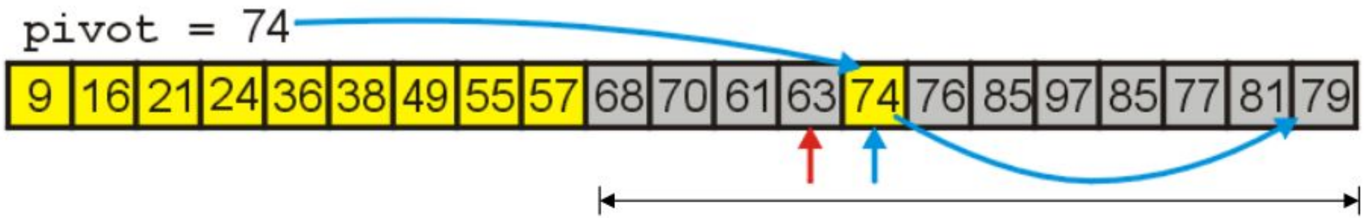
24	49	16	38	55	21	36	9	57	68	81	85	63	79	74	85	97	61	77	70	76

9	21	16	24	55	49	36	38	57	68	81	85	63	79	74	85	97	61	77	70	76

9	16	21	24	55	49	36	38	57	68	81	85	63	79	74	85	97	61	77	70	76

9	16	21	24	36	38	49	55	57	68	81	85	63	79	74	85	97	61	77	70	76

QuickSort - exemplo



.
.
.

Quicksort - Implementação

```c
void quicksort(int *item, int left, int right) {
  int i, j;
  int x, y;
  i = left;
  j = right;
  x = item[(left + right) / 2]; /* elemento pivo */
  /* partição das listas */
  do {
    /* procura elementos maiores que o pivô na primeira parte*/
    while (item[i] < x && i < right) {
      i++;
    }
      /* procura elementos menores que o pivô na segunda parte */
    while (x < item[j] && j > left) {
      j--;
    }
    if (i <= j) {
      /* processo de troca (ordenação) */
      y = item[i];
      item[i] = item[j];
      item[j] = y;
      i++;
      j--;
    }
  } while (i <= j);
  /* chamada recursiva */
  if (left < j) {
    quicksort(item, left, j);
  }
  if (i < right) {
    quicksort(item, i, right);
  }
}
```

https://scanuto.com/quicksort.c

Quicksort - Observações

- Apesar de possuir complexidade $O(n^2)$ no pior caso (quando o procedimento de particionamento produz um subproblema com n -1 elementos e um com 0 elementos), o QuickSort é considerado um dos melhores algoritmos de ordenação.
- Isso se deve à sua eficiência no melhor e no caso médio $O(n \log n)$.
- QuickSort **não** é um algoritmo estável.

Ordenação em tempo linear

- Algoritmos de ordenação por **comparação**:
  - *Quicksort;*
  - *MergeSort*;
  - ...
- Possuem, no melhor caso, complexidade O(*n lg n*);
- Podem existir algoritmos melhores?

Ordenação em tempo linear

- A resposta é **SIM**, desde que:
  - A entrada possua características especiais;
  - Algumas restrições sejam respeitadas;
  - O algoritmo não seja puramente baseado em comparações;
  - A implementação seja feita da maneira adequada.
- Tempo linear: $O(n)$;
- Algoritmos:
  - Ordenação por contagem (*Counting sort*);
  - *Radix sort*;
  - *Bucket sort*.

ORDENAÇÃO EM TEMPO LINEAR

Ordenação por contagem

- Pressupõe que cada elemento da entrada é um inteiro na faixa de *0* a *k*, para algum inteiro *k*;
- Idéia básica:
  - Determinar para cada elemento da entrada *x* o número de elementos maiores que *x*;
  - Com esta informação, determinar a posição de cada elemento
    - Ex.: Se 17 elementos forem menores que *x* então *x* ocupa a posição de saída 18.

Ordenação por contagem

- Algoritmo:
  - Assumimos que o vetor de entrada é *A[1,...,n];*
  - Outros dois vetores são utilizados:
    - *B[1,...,n]* – armazena a saída ordenada;
    - *C[1,...,k]* – é utilizado para armazenamento temporário.

Ordenação por contagem

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] down to 1
        do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	**2**	**5**	**3**	**0**	**2**	**3**	**0**	**3**

k = 5

	0	1	2	3	4	5
C						

	1	2	3	4	5	6	7	8
B								

Ordenação por contagem

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] down to 1
        do B[C[A[j]]] ← A[j]
              C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k = 5

	0	1	2	3	4	5
C	0	0	0	0	0	0

	1	2	3	4	5	6	7	8
B								

Ordenação por contagem

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] down to 1
        do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k = 5

	0	1	2	3	4	5
C	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
B								

Ordenação por contagem

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k = 5

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

Ordenação por contagem

Onde colocar o elemento da posição 8 em A no vetor B?

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
           C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k = 5

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

Ordenação por contagem

Onde colocar o elemento da posição 8 em A no vetor B?

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] downto 1
        do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k = 5

	0	1	2	3	4	5
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B								

Ordenação por contagem

Onde colocar o elemento da posição 8 em A no vetor B?

```
for i ← 0 to k
      do C[i] ← 0
for j ← 1 to length[A]
      do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
      do C[i] ← C[i] + C[i −1]
for j ← length[A] downto 1
      do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	**2**	**5**	**3**	**0**	**2**	**3**	**0**	**3**

k = 5

	0	1	2	3	4	5
C	**2**	**2**	**4**	**7**	**7**	**8**

	1	2	3	4	5	6	7	8
B							3	

Ordenação por contagem

```
for i ← 0 to k
        do C[i] ← 0
for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
        do C[i] ← C[i] + C[i −1]
for j ← length[A] down to 1
        do B[C[A[j]]] ← A[j]
            C[A[j]] ← C[A[j]] −1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

$k = 5$

	0	1	2	3	4	5
C	0	2	2	4	7	7

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Ordenação por contagem

- O tempo de execução é dado em função do valor de k;
- Roda em tempo O(n + k);
- Se tivermos k = O(n), então o algoritmo executa em tempo O(n);

Radix Sort

- Pressupõe que as chaves de entrada possuem limite no valor e no tamanho (quantidade de dígitos);
- Ordena em função dos dígitos (um de cada vez):
  - A partir do mais significativo;
  - Ou a partir do menos significativo?
- É essencial utilizar um segundo algoritmo estável para realizar a ordenação de cada dígito.

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

3 2 9
4 5 7
6 5 7
8 3 9
4 3 6
7 2 0
3 5 5

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

7 2 0
3 5 5
4 3 6
4 5 7
6 5 7
3 2 9
8 3 9

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

7	2	0
3	5	5
4	3	6
4	5	7
6	5	7
3	2	9
8	3	9

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

3 2 9	7 2 0	7 2 0
4 5 7	3 5 5	3 2 9
6 5 7	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9
4 3 6	6 5 7	3 5 5
7 2 0	3 2 9	4 5 7
3 5 5	8 3 9	6 5 7

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

$\rightarrow$

7	2	0
3	5	5
4	3	6
4	5	7
6	5	7
3	2	9
8	3	9

$\rightarrow$

7	2	0
3	2	9
4	3	6
8	3	9
3	5	5
4	5	7
6	5	7

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3 9	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:

Como ficaria a partir o dígito mais significativo?

3 2 9	→	7 2 0	→	7 2 0	→	3 2 9
4 5 7		3 5 5		3 2 9		3 5 5
6 5 7		4 3 6		4 3 6		4 3 6
8 3 9		4 5 7		8 3 9		4 5 7
4 3 6		6 5 7		3 5 5		6 5 7
7 2 0		3 2 9		4 5 7		7 2 0
3 5 5		8 3 9		6 5 7		8 3 9

Radix Sort – Funcionamento

- A partir dos dígitos menos significativos:
  E se a ordenação não fosse estável?

3 2 9	7 2 0	7 2 0	3 2 9
4 5 7	3 5 5	3 2 9	3 5 5
6 5 7	4 3 6	4 3 6	4 3 6
8 3 9	4 5 7	8 3	4 5 7
4 3 6	6 5 7	3 5 5	6 5 7
7 2 0	3 2 9	4 5 7	7 2 0
3 5 5	8 3 9	6 5 7	8 3 9

Radix Sort – Pseudo Código

- Como dito anteriormente, o *Radix Sort* consiste em usar um outro método de ordenação (estável) para ordenar as chaves em relação a cada dígito;
- O código, portanto, é muito simples:

```
for i ← 1 to d
    utilize um algoritmo estável para ordenar o
    array A pelo i-ésimo dígito.
```

- Onde:
  - *d* é número de dígitos;
  - *A* é o *array* de entrada.

Bucket Sort

- Assume que a entrada consiste em elementos distribuídos de forma **<u>uniforme</u>** sobre o intervalo *[0,1)*;
- A idéia do *Bucket Sort* é dividir o intervalo *[0,1)* em *n* subintervalos de mesmo tamanho (*baldes*), e então distribuir os *n* números nos *baldes*;
- Uma vez que as entradas são uniformemente distribuídas não se espera que muitos números caiam em cada *balde*;

Bucket Sort

- Para produzir a saída ordenada, basta ordenar os números em cada *balde*, e depois examinar os *baldes* em ordem, listando seus elementos;
- A função para determinação do índice do *balde* correto é $\lfloor n \times A[i] \rfloor$
- Vamos a um exemplo com 10 números
  - *A* é o *array* de entrada;
  - *B* é o *array* com os baldes.

Bucket Sort – Funcionamento

B

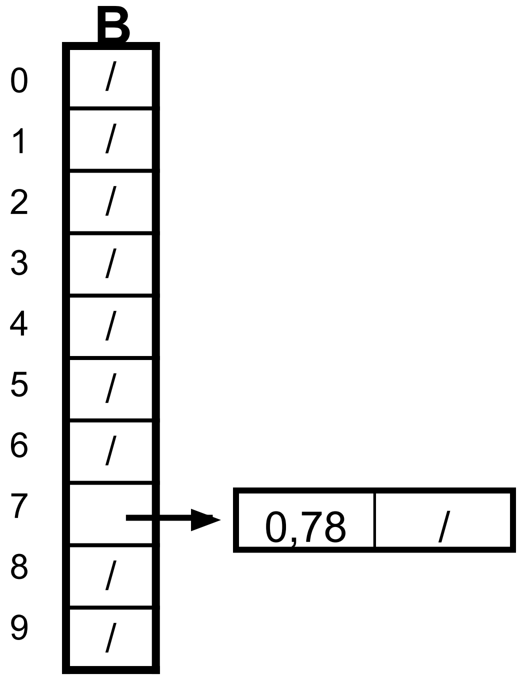
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

10 x 0,78 = **7**,8

A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68

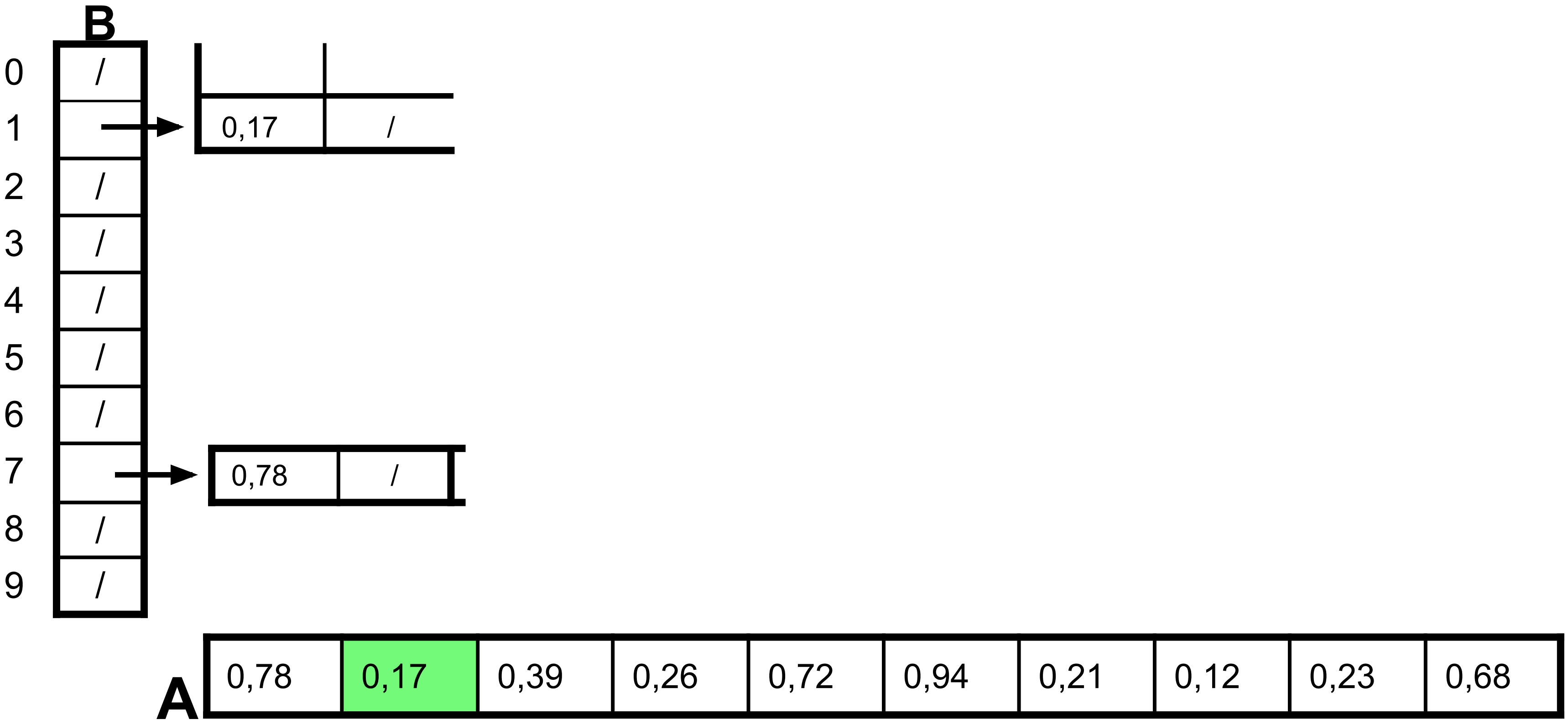
Bucket Sort – Funcionamento



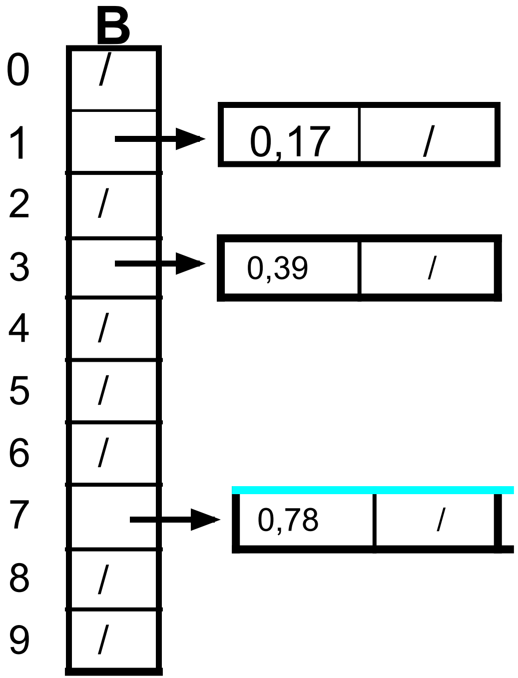
A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68

Bucket Sort – Funcionamento



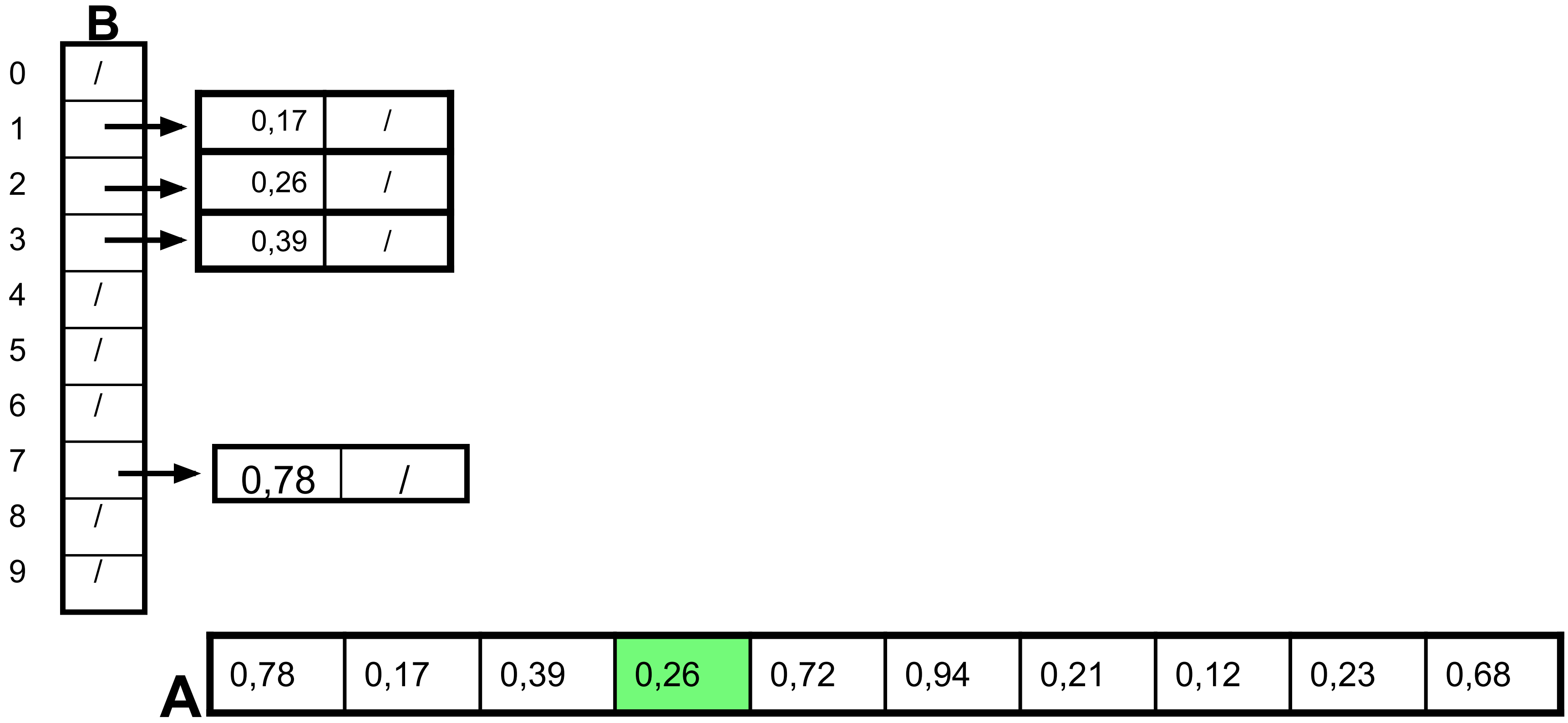
Bucket Sort – Funcionamento



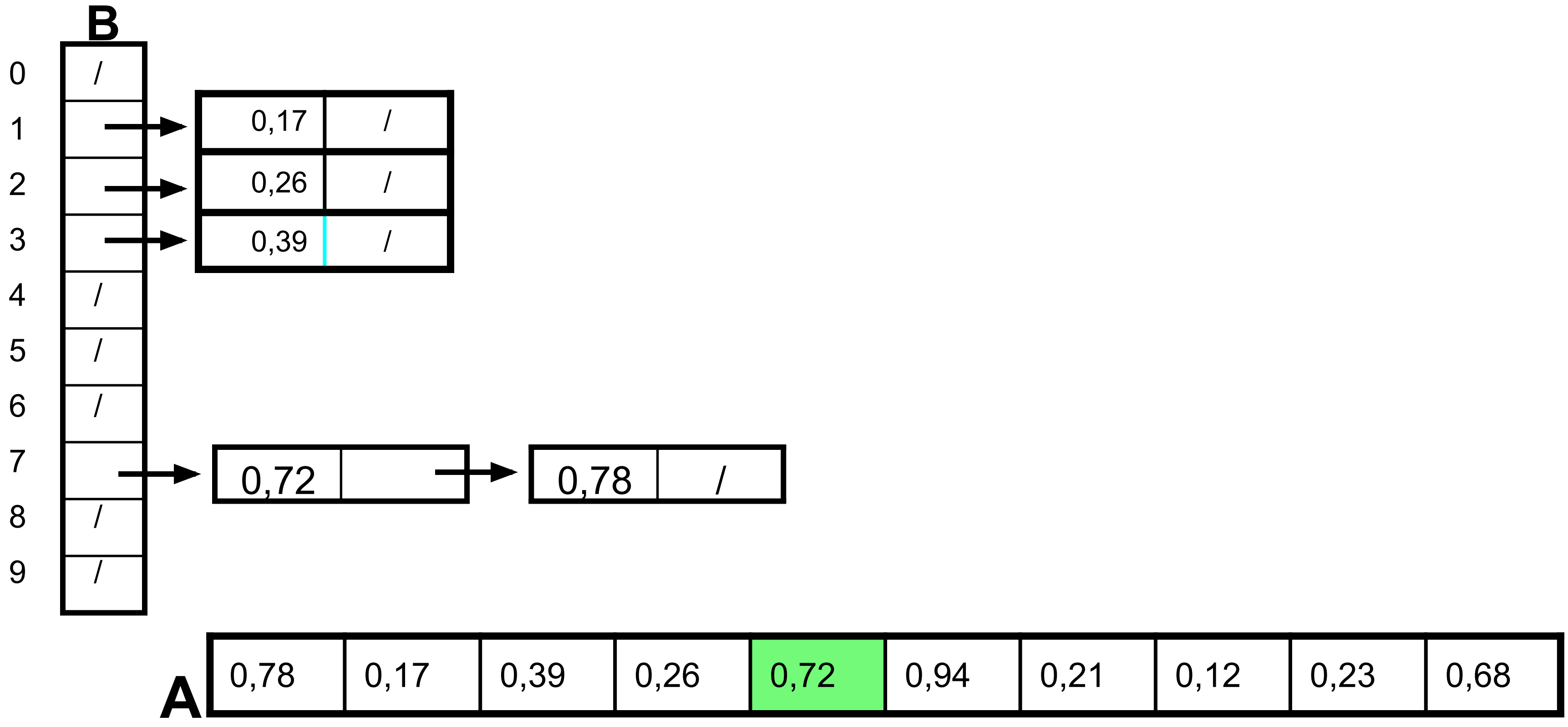
A

0,78	0,17	0,39	0,26	0,72	0,94	0,21	0,12	0,23	0,68

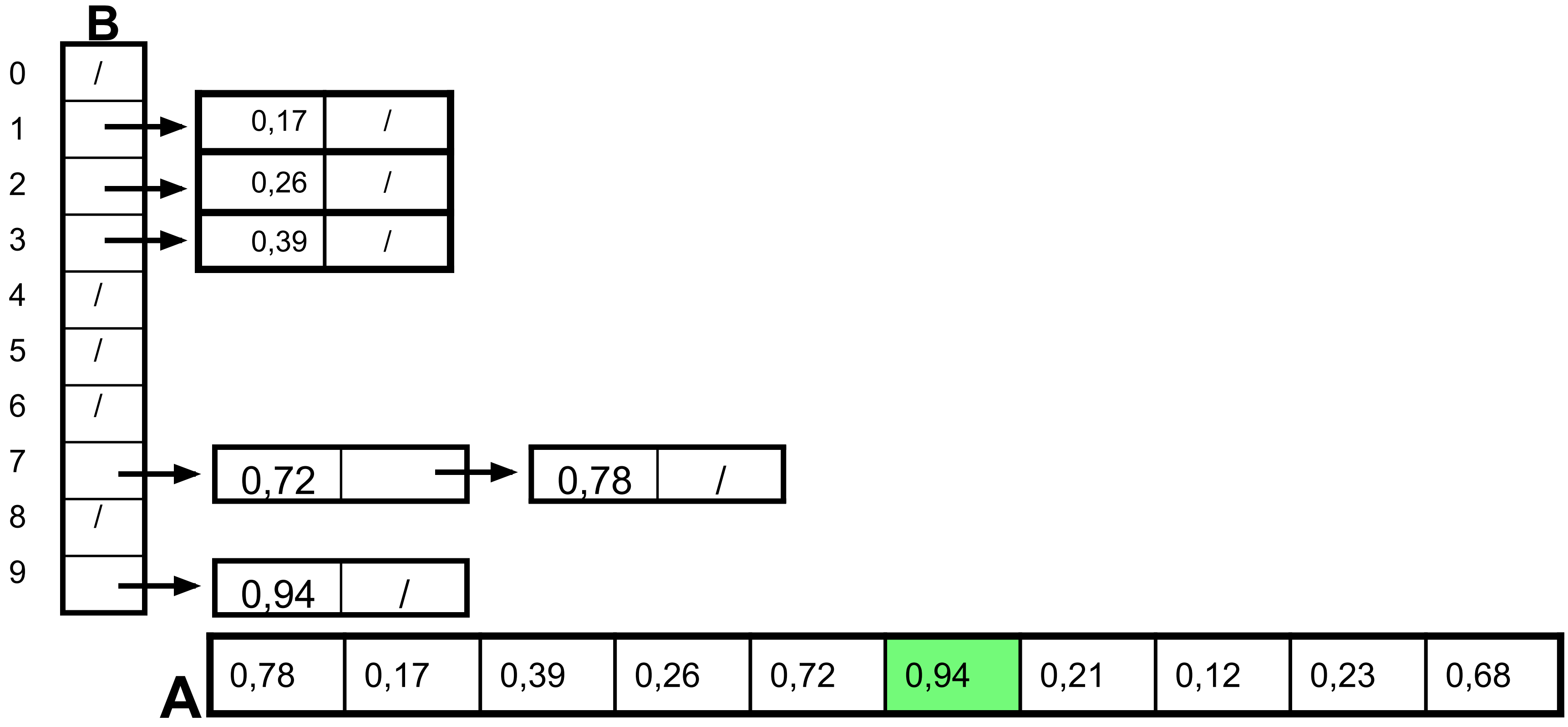
Bucket Sort – Funcionamento



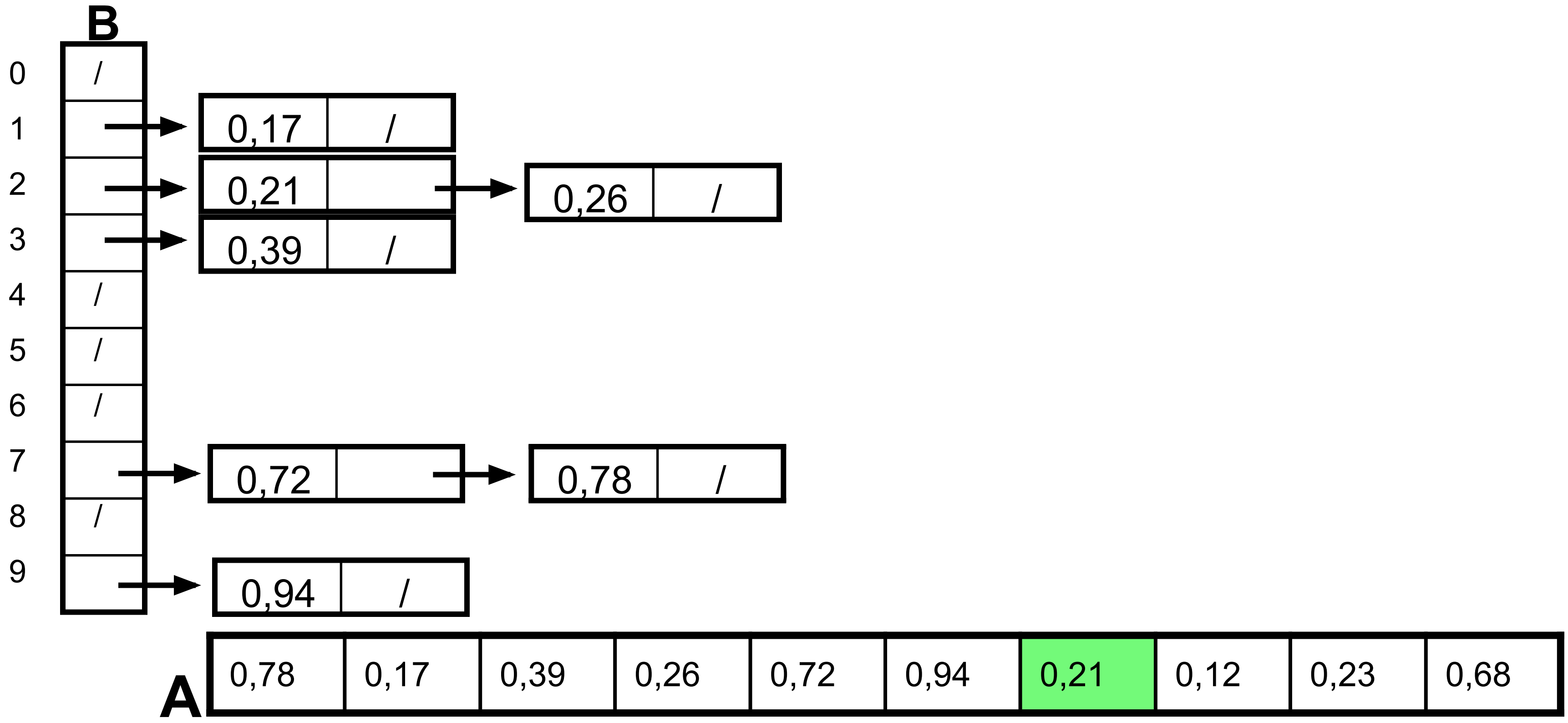
Bucket Sort – Funcionamento



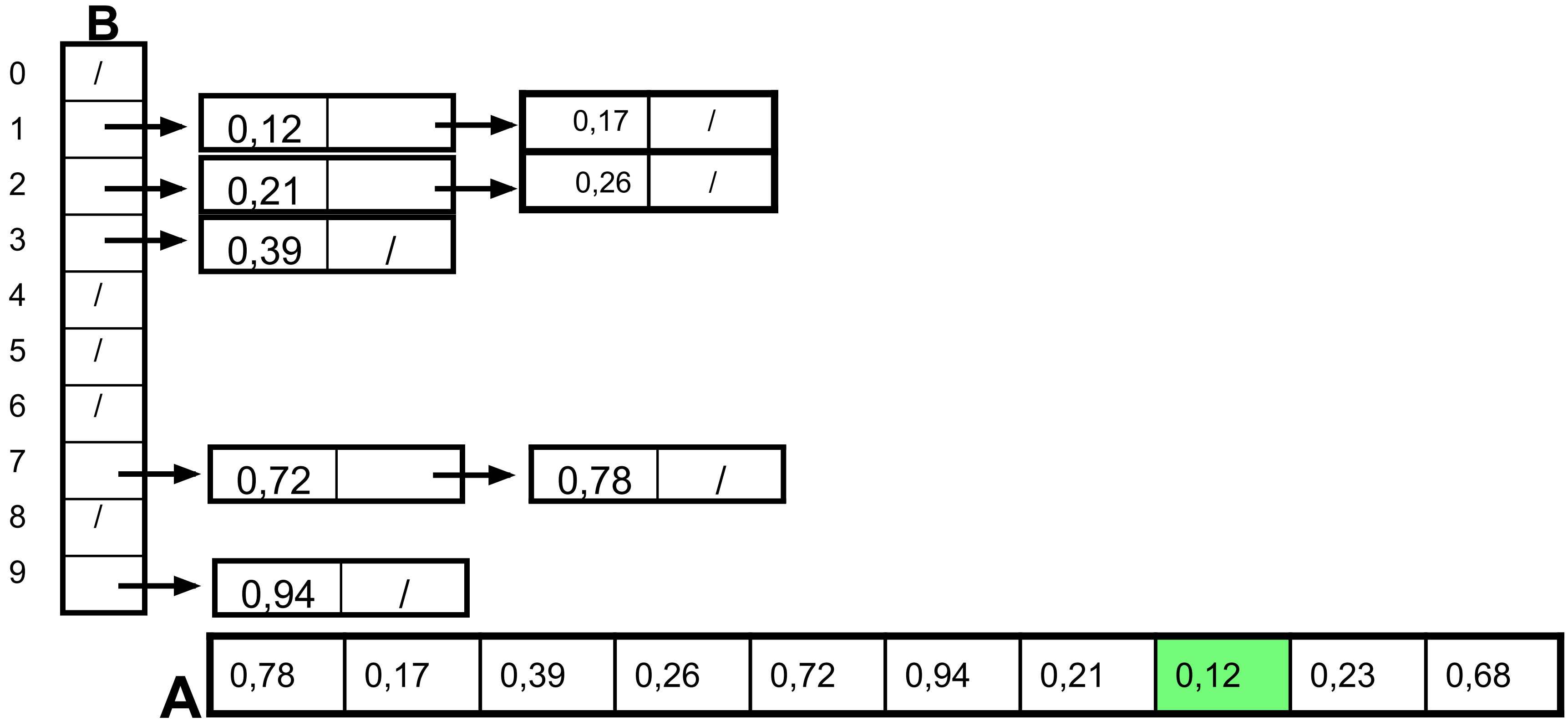
Bucket Sort – Funcionamento



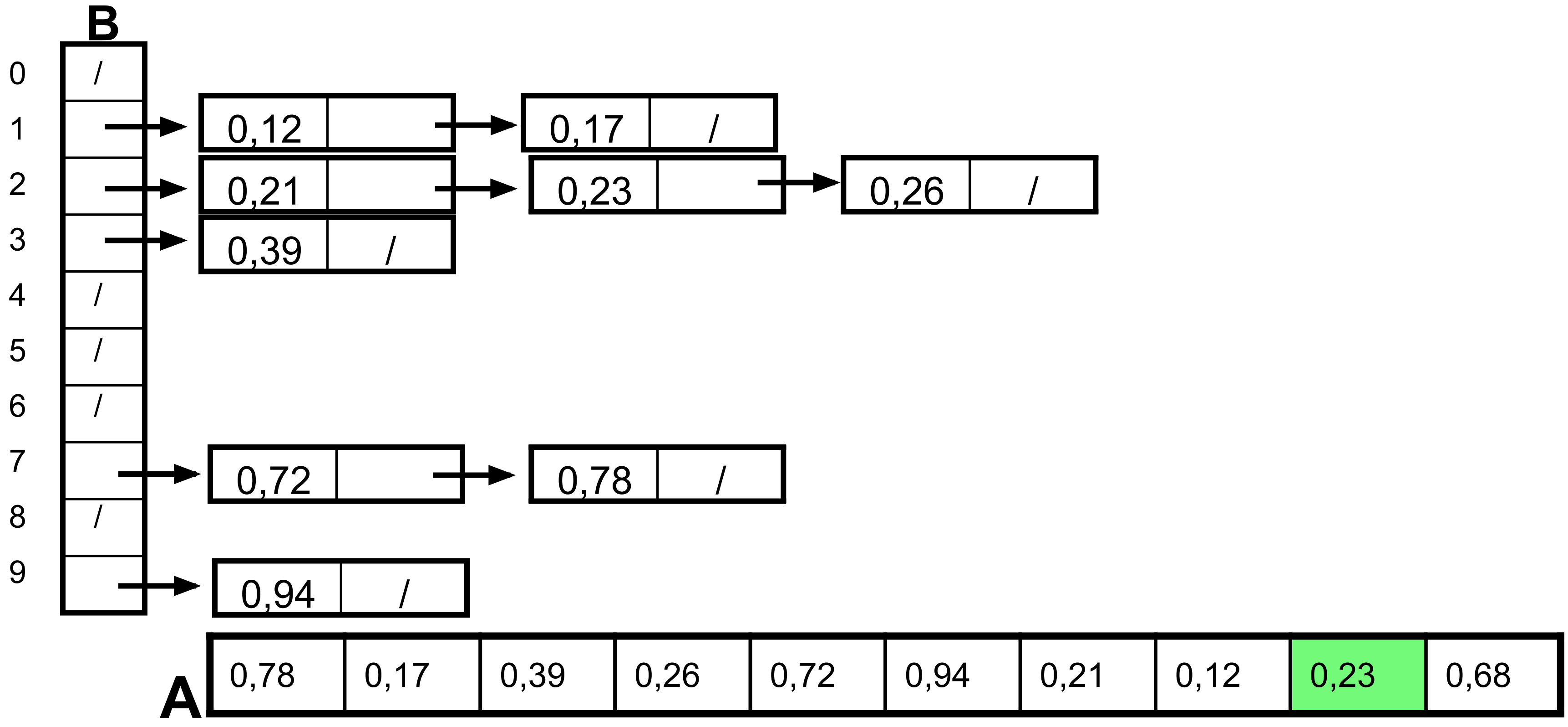
Bucket Sort – Funcionamento



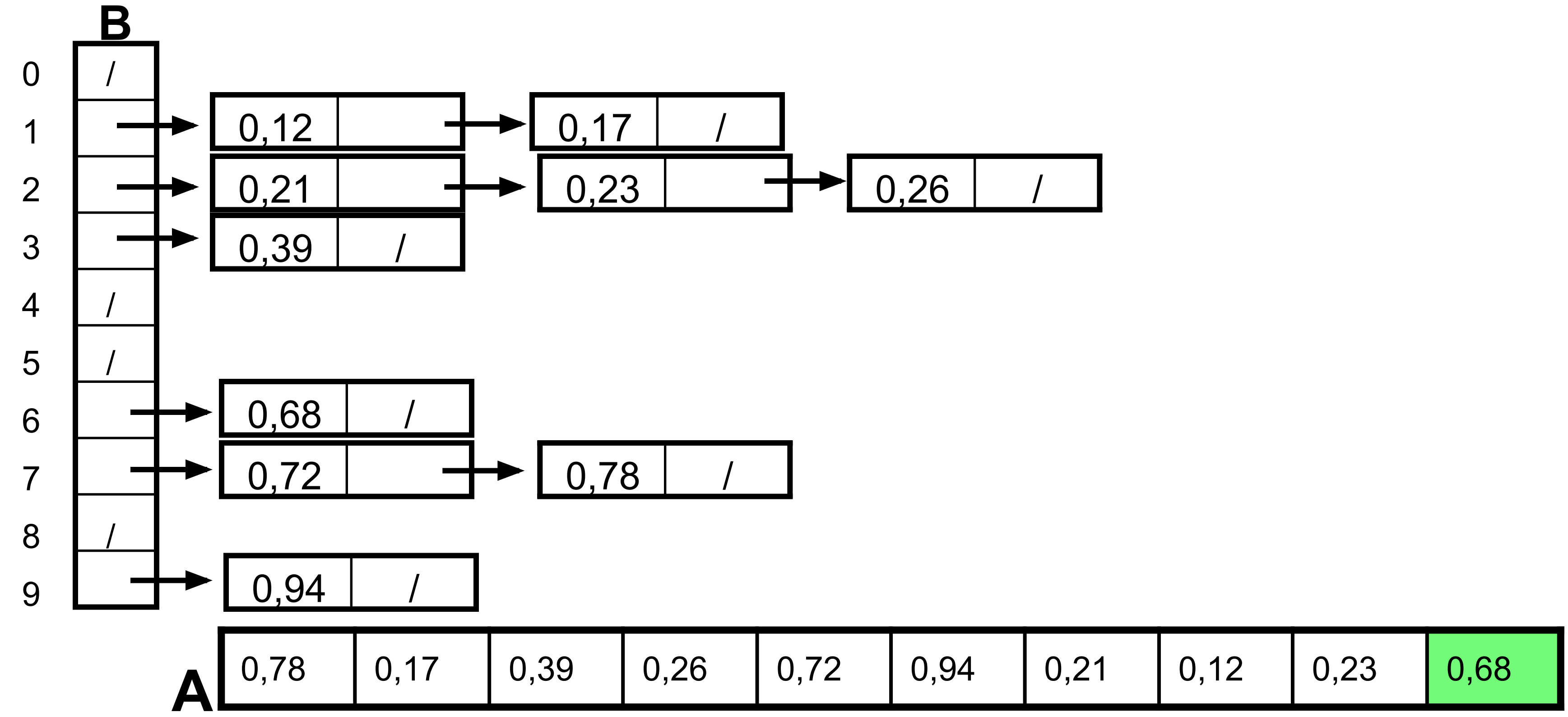
Bucket Sort – Funcionamento



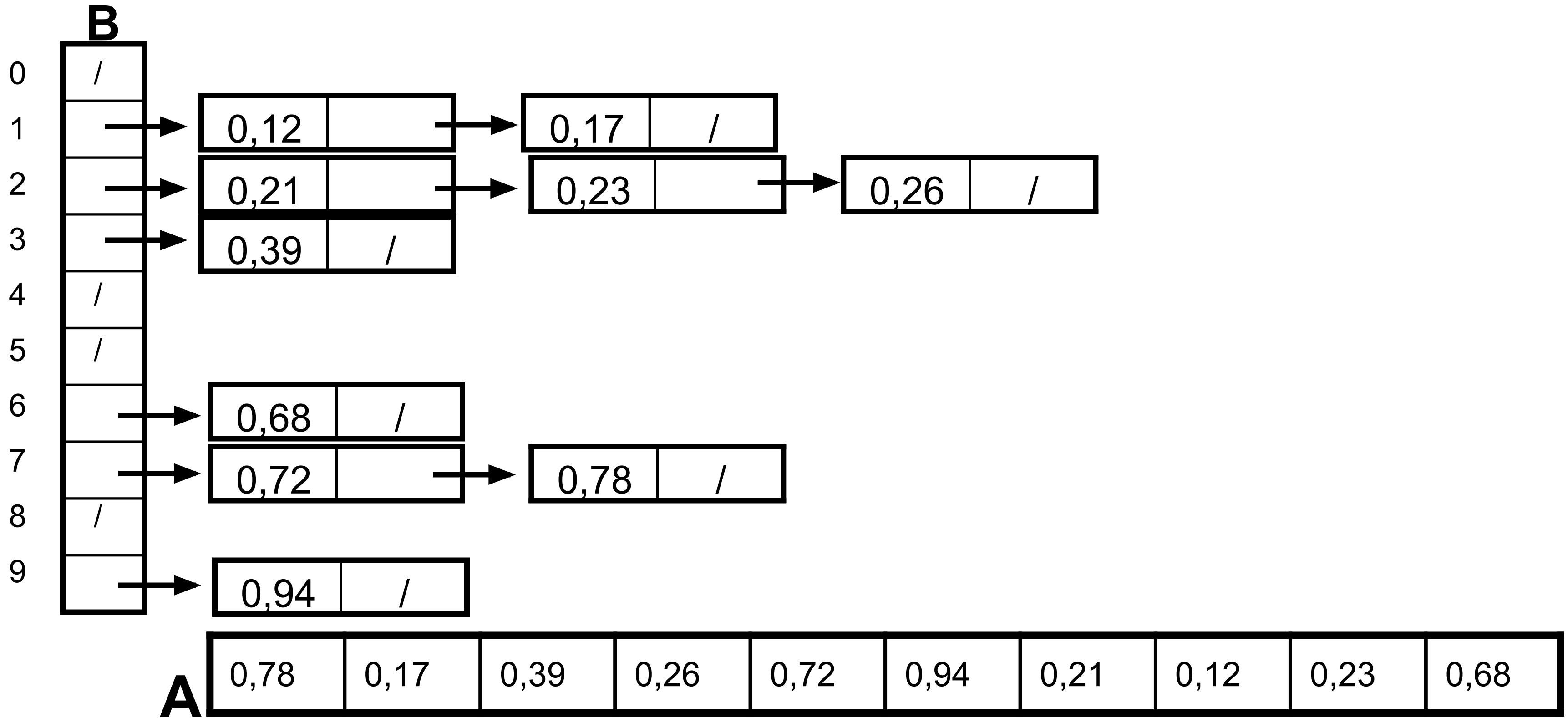
Bucket Sort – Funcionamento



Bucket Sort – Funcionamento



Bucket Sort – Funcionamento



Conclusões - Ordenação em tempo Linear

- Foram vistos três algoritmos de ordenação linear (tempo O(*n*)). Que são então melhores que os algoritmos de ordenação por comparação (tempo O(*n lg n*));

- **Entretanto**, nem sempre é interessante utilizar um destes três algoritmos:
  - Todos eles pressupõem algo sobre os dados de entrada a serem ordenados.

Referências

Estrutura de Dados descomplicada em Linguagem C (André Backes): Cap 3;

M. LAUREANO, Estrutura de Dados com Algoritmos e C, Brasport, 2008. (seção 8.4)

Atividade

Faça uma análise empírica dos algoritmos mergesort, quicksort, selectionsort e countsort

Para a análise empírica:

1. Gere cinco vetores de tamanho 10000 com número inteiros (de 0 a 65535) aleatórios utilizando a função rand()
2. Execute cada um dos algoritmos (ex. mergesort) utilizando cada vetor gerado no passo (1).
   a. Meça o tempo de cada uma das 5 execuções para cada algoritmo de ordenação
   b. Meça o número de operações de comparação feitas por cada algoritmo.
3. Compute o tempo médio das 5 execuções para cada algoritmo e apresente os resultados. Explique o porquê dos resultados obtidos.